



TYPESCRIPT @BING

KIRAN BADAM & SARVESH NAGPAL

Agenda



Why we chose Type Script ?

- The sweet spot between Types and Idiomatic Javascript



Current adoption

- 15% current adoption.
- Target: 100%



Integration with Bing Resource Management System (RMS)

- Save and Refresh
- Build



Stint with AMD

- How we are trying to move to AMD with Typescript



JS2TS Magic

- Our solution for auto converting Javascript files to TypeScript files



Why we chose Type Script ?

We love writing idiomatic Javascript

- Module pattern
- Patterns like currying

We also realize the lack of tooling to help maintainability

- Lack of strongly typed contracts
- Lack of IDE support with intellisense, refactoring etc.

Typescript hits the sweet spot

- Compared to other technologies like Script#

Current Adoption

15% of our Javascript Codebase is in TypeScript

- It is growing at a very fast rate

25,000 lines of code so far in Type Script.

- Bing Mobile Experiences
- Bing Windows 8 App
- Friends Photos (Social)
- Parts of Search Results Page

Target is to hit 100% adoption

- JS2TS tool that Sarvesh will demo will help us adopt TypeScript at a scale.

Best Practices

<http://bing/wiki/TypeScript>

Favor modules over classes

- Most of Bing Web Page code is asynchronous, reactive and pub-sub based.
- Classes are verbose and at Bing, page weight is a significant concern.

Code to Interfaces

Integration with Bing RMS

- ▶ RMS manages both the compilation (crunching, bundling etc) and the rendering of Javascript files (inlined, referenced etc).
- ▶ We added support for TypeScript files into RMS
 - ▶ When a typescript file changes
 - ▶ RMS invokes TS compiler to compile the file along with its dependencies
 - ▶ The generated file is then run through a cruncher
 - ▶ And then is concatenated with other files if necessary.
 - ▶ We use Node JS to invoke TypeScript compiler (`<Import Project="$(PackagesRoot)\TypeScript.NodeJs.Tools\TypeScript.NodeJs.AMD.targets" />`)
- ▶ Batch compilation saves time by orders of magnitude.

Stint With AMD

[Js_AjaxHistory]

```
_meta.type=Microsoft.Search.Frontend.ResourceManagement.Configuration.IResourceConfig  
ResourceKey=Ajax.History  
RenderEndpoint=JsAfterOnLoad  
ReferencingMethod=alwaysReference  
FilePath&win8:app2=Scripts\Ajax\ajax.pushstate.js  
FilePath=Scripts\Ajax\ajax.history.js  
Dependencies=Shared.Event.Customs|
```


Stint With AMD

- ▶ Ajax.History is the interface (d.ts)
- ▶ Ajax.PushState and Ajax.History are two possible implementations of the above interface
- ▶ Developers code to the d.ts
 - ▶ Import("Ajax.History");
 - ▶ AMD Loader will at runtime decide which implementation to load, based on its configuration.
- ▶ Challenges:
 - ▶ Modules and Interfaces are incompatible
 - ▶ Importing a d.ts file is not allowed.

JS2TS Demo



CloudMake – Automating Builds using Typescript

Tools for Software Engineers
Wolfram Schulte

<http://tse> & <http://codebox/clouddev>

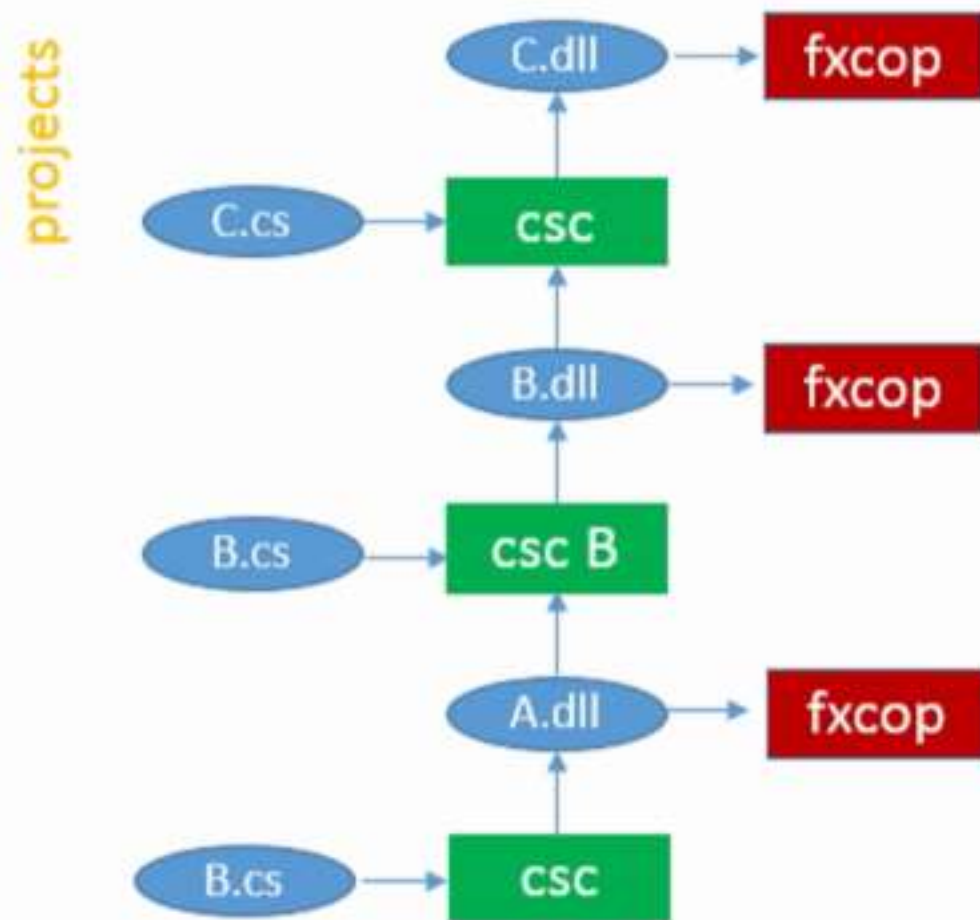
TypeScript Meetup 6/6/2013

Why do we need another language for builds, installers, tests, deployments?

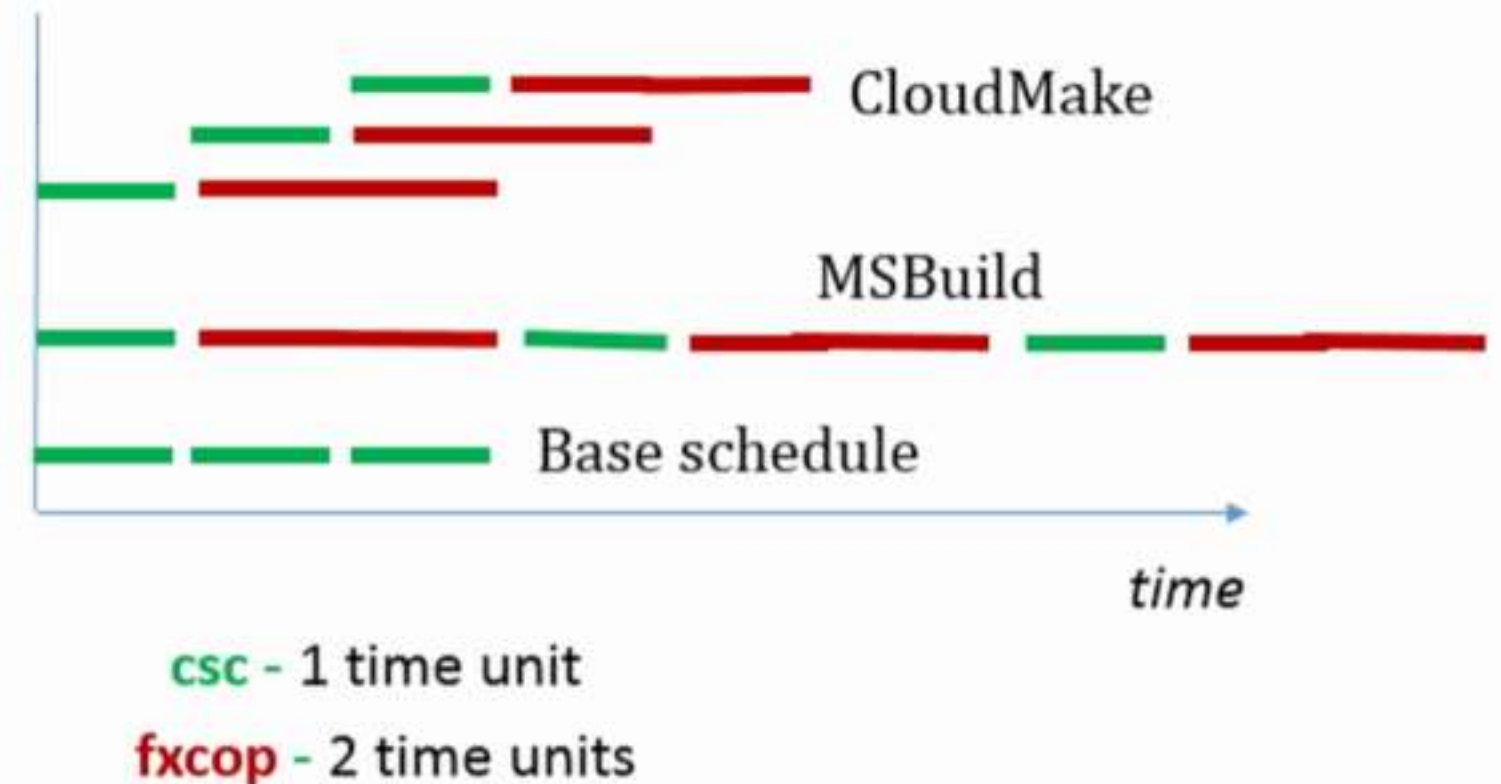
- Reliability
 - Comprehensibility, repeatability, precise dependency and constraint manageability
- Composability
 - Allow building abstractions: functions, modules, versions, visibility
- Speed
 - Response time proportional to size of change, never wait longer than longest critical path
- Resource efficiency
 - Reuse your earlier work or work done by others, scale-up and out, multi-tenancy

Finding the Right Level of Abstraction

Example: MSBuild after Target



Schedules



CloudMake

A language and runtime system for **describing and executing automated workflows**, like builds, installers, tests, or deployments.

CloudMake

- is designed to be **compatible with Typescript** as much as possible
- avoids features that are difficult to control
- is **easy to extend** with external tools
- **works without** a sophisticated programming **environment**
- is the language of choice for CloudDev!

CloudMake – Hello World

```
///<referencepath= ' //Tools/StandardTools.cm'>
```

```
var main      =      csc ({sources:[ 'helloworld.cs' ]});
```



Output Artifact

Tool/Function

Input Artifact

Multi-directory Builds

```
///<referencepath= 'Speller/Speller.cm'>
```

```
module Microsoft.Office.Word {
```

```
    export var EditingSurface = csc(  
        {
```

```
            AssemblyName: "Microsoft.Office.Word.EditingSurface",
```

```
            OutputType: OutputType.Library,
```

```
            Sources: ['Canvas.cs', 'Renderer.cs', 'MessageLoop.cs'],
```

```
            References: [Microsoft.Office.Word.Speller.CoreSpeller.dll]
```

```
        });
```

```
    }
```

Reference of other build artifact

Everything typed, no type annotation needed

Composability

```
module Microsoft.Office.Word.Speller {  
  export var CoreSpeller = csc(  
    {  
      AssemblyName: "Microsoft.Office.Word.Speller",  
      Sources: ['language.cs', 'grammar.cs'],  
      ...  
    });  
  
  export var En = BuildLanguage("En")  
  export var Nl = BuildLanguage("Nl")  
  ...  
  
  function BuildLanguage(languageCode) {  
    return csc(  
      {  
        AssemblyName: "Microsoft.Office.Word.Speller.Languages." + languageCode,  
        Sources: [Path.combine('.', languageCode + ".cs")],  
        References: [CoreSpeller.dll]  
      });  
  }  
}
```


Defining Tools

```
interface CscArgs {
    AssemblyName? : String;
    Sources        : CsFile[];
    References?    : Assembly[];
    OutputType?    : OutputType;
}

function csc (args: CscArgs) : Assembly
{
    var output = Path.Combine('//output/', args.assemblyName, ".dll")
    var srcs   = sources.map(s => s.cs)
    var refs   = references.map(r => r.dll)

    var cmdargs = ["/out:" + outputAssembly]
                  .add ("/target:" + ToTargetString(OutputType))
                  .addRange (refs.map( r => "/r:" + r))
                  .addRange (srcs)

    return {dll: exec({
        tool           : '//clr/csc.exe'
        arguments      : cmdargs,
        dependencies    : srcs.AddRange(refs),
        expectedResults : [output] }) [0] }
}
```

CloudMake

- A Purely Functional Subset of TypeScript

Inherits Syntax, Type System and subset of semantics

- **Primitive types** and operations are the *same*
- **Arrays and Objects** support *only non-side effecting operations*
- **All variables** (except for function locals) are *single assignment only*
- **No support for *classes, overloading***

Special support for

- **Paths**, written with '...' and, strings "..."
- **Variable initialization**, which is **lazy**, i.e. CM is more defined than TS

CloudMake – Implementation and Performance

Implemented as **direct interpreter in .NET**

- Parse tree used for evaluation, type tagged representation
- Two phase execution: (1) evaluates the prog to a schedule (2) run the schedule

Performance (exemplary)

Evaluating CM	7X faster than compiling Typescript
Sequential CM	1.0 – 1.2X faster than running Sequential MS Builds
Multicore CM	1.1 – 2X faster than running Multicore MS Builds
Incremental CM	1.1 to 25X faster than running Incremental MS Builds
Distributed CM	Worst case execution time is duration of critical path

CloudMake - Next steps

- Work with TypeScript team on language features and tool optimizations
- Add support for **multi-flavored and multi-versioned builds**
- Synthesizing CloudMake **programs from traces**
- Create **simple exe** so that **you can use it**
- Get **your feedback**, and improve!

Overview

- Decorators
- Iterators and for..of
- Generators
- Async/Await

Example: TypeUnit

```
import Promise = module('promise');
import TypeUnit = module('typeunit');
var test = TypeUnit.test,
    timeout = TypeUnit.timeout,
    data = TypeUnit.data;

export class PromiseFixture {
  @test
  @timeout(100)
  computeAsync(): void {
    ...
    p.then(
      @stage function(v) { TypeUnit.Assert.equal(1, v); },
      @stage function(e) { TypeUnit.Assert.fail(e); });
  }

  static parseData(): any[][] { ... }

  @test
  @data(PromiseFixture.parseData)
  parse(url, match): void { ... }
}
```

- `@test` decorator to mark test methods
- `@timeout` decorator marks maximum duration for test
- `@data` decorator performs data-bound tests
- `@stage` decorator wraps a callback in the test framework so that failed assertions are caught
- TypeUnit is available at <http://codebox/typeunit>

Example: Ember.js to TypeScript (with Decorators)

ES5

```
App.Person = DS.Model.extend({
  firstName: DS.attr('string'),
  lastName: DS.attr('string'),
  birthday: DS.attr('date'),
  fullName: function() {
    return this.get('firstName') +
      ' ' +
      this.get('lastName');
  }.property('firstName', 'lastName')
});
```

TypeScript

```
class Person extends DS.Model {
  // replaces firstName with a get/set accessor
  @attr('string')
  firstName: string;

  @attr('string')
  lastName: string;

  @attr('date')
  birthday: Date;

  @property('firstName', 'lastName')
  get fullName() {
    return this.firstName +
      ' ' +
      this.lastName;
  }
};
```

Example: TypeUnit

```
import Promise = module('promise');
import TypeUnit = module('typeunit');
var test = TypeUnit.test,
    timeout = TypeUnit.timeout,
    data = TypeUnit.data;

export class PromiseFixture {
  @test
  @timeout(100)
  computeAsync(): void {
    ...
    p.then(
      @stage function(v) { TypeUnit.Assert.equal(1, v); },
      @stage function(e) { TypeUnit.Assert.fail(e); });
  }

  static parseData(): any[][] { ... }

  @test
  @data(PromiseFixture.parseData)
  parse(url, match): void { ... }
}
```

- `@test` decorator to mark test methods
- `@timeout` decorator marks maximum duration for test
- `@data` decorator performs data-bound tests
- `@stage` decorator wraps a callback in the test framework so that failed assertions are caught
- TypeUnit is available at <http://codebox/typeunit>

Iterators

```
/// <reference path="generators.ts" />
function* filter(
  iter: Iterator,
  predicate: (value: any) => bool): Iterator {

  for(var value of iter) {
    if (predicate(value)) {
      yield value;
    }
  }
}

var a = [1, 2, 3];
var f = filter(a, v => v % 2);
for(var v of f) {
  console.log(v); // LOG: 1
                  // LOG: 3
}

// class examples
class Map {
  *keys(): Iterator { ... }
  *values(): Iterator { ... }
  __iterator__: Iterator { ... }
}
```

- ``function*`` signals rewrite of function body
- ``for...of`` is rewritten to call ``__iterator__`` and repeatedly execute calls to ``next``
- Using Iterators/generators requires a reference to or an import of the `'generators'` module defined in `generators.ts`

Generators

(based on TS 0.8.3 and ES6 spec prior to May, 2013)

```
/// <reference path="generators.ts" />
function* backAndForth() {
    var b = yield 3;
    console.log("b = %s", b);

    var d = yield b + 1;
    console.log("d = %s", d);
    return d - 1;
}

var o = backAndForth();
try {
    var a = o.next();
    console.log("a = %s", a);

    var c = o.send(a + 2);
    console.log("c = %s", c);

    o.send(c * 2);
}
catch (ex) {
    if (!isStopIteration(ex)) { throw e; }
    var e = ex.value;
    console.log("e = %s", e);
}
```

- **Output:**
LOG: a = 3
LOG: b = 5
LOG: c = 6
LOG: d = 12
LOG: e = 11
- ``function*`` signals rewrite of function body, allowing ``yield`` to be interpreted as a keyword and not an identifier.
- The ES6 ``yield`` keyword is an expression, allowing callers to send values back into the generator as the result of the expression.
- The ``return`` statement in a generator throws ``StopIteration`` with an optional value.
- Using Iterators/generators requires an reference to or an import of the `'generators'` module defined in `generators.ts`

Async/Await

```
/// <reference path="generators.ts" />
import futures = module('futures');
import httpclient = module('httpclient');

async function getJSONAsync(url: string): futures.Future<any> {
    var client = new httpclient.HttpClient();
    var response = await client.GetAsync(url);
    return JSON.parse(response.responseText);
}

async function fetchAuthProviders(vm: AuthViewModel): void {
    var providers = await getJSONAsync(authProviderUrl);
    vm.providers(providers);
}
```

- ``async`` keyword signals rewrite of function body
- Return type annotation must be a ``Thenable`` or ``void``
- ``void`` return type annotation signals that any errors are thrown to the browser/engine
- ``await`` keyword halts execution until the thenable produces a value or a result.
- ``await`` auto-lifts non-thenable values
- Using `async/await` requires an import of the `'generators'` module defined in `generators.ts`

Notes

Current Version

- Based on TS 0.8.3
- Supports:
 - Decorators (functions; classes; class methods, getters, setters)
 - Iterators (pre-May, 2013)
 - Generators (pre-May, 2013)
 - Async/Await
 - VS Editor
- Not Supported:
 - Decorators (fields)
- Fork available [here](#)

New Version

- Based on TS 0.9.0
- Supports:
 - Decorators (functions; classes; class methods, getters, and setters)
- Coming Soon:
 - Decorators (fields; object literal methods, getters, and setters; better type-safety)
 - Iterators (post-May, 2013)
 - Generators (post-May, 2013)
 - Async/Await
 - VS Editor
- Long-term:
 - Array/Object destructuring
 - Generator/Array comprehensions
- Fork Available [here](#)

Async/Await

```
/// <reference path="generators.ts" />
import futures = module('futures');
import httpclient = module('httpclient');

async function getJSONAsync(url: string): futures.Future<any> {
    var client = new httpclient.HttpClient();
    var response = await client.GetAsync(url);
    return JSON.parse(response.responseText);
}

async function fetchAuthProviders(vm: AuthViewModel): void {
    var providers = await getJSONAsync(authProviderUrl);
    vm.providers(providers);
}
```

- ``async`` keyword signals rewrite of function body
- Return type annotation must be a ``Thenable`` or ``void``
- ``void`` return type annotation signals that any errors are thrown to the browser/engine
- ``await`` keyword halts execution until the thenable produces a value or a result.
- ``await`` auto-lifts non-thenable values
- Using `async/await` requires an import of the `'generators'` module defined in `generators.ts`

Async/Await Definitions

```
interface Thenable {  
  then(resolve: Function, reject?: Function): any;  
}
```

```
interface Awaiter {  
  awaiter: Thenable;  
}
```

```
interface Deferred {  
  resolve(value: any): void;  
  reject(value: any): void;  
  promise: Thenable;  
}
```

```
declare var Awaiter: {  
  defer(): Deferred;  
}
```

- Defined in '**generators**' module
- General interface designed to match Promise/A-style promises/futures.
- Uses a built-in general purpose Promise/A-like object internally.
- Developers can replace built-in behavior by overriding the ``Awaiter.defer`` method with a compatible API (or wrapper).

Async/Await

```
/// <reference path="generators.ts" />
import futures = module('futures');
import httpclient = module('httpclient');

async function getJSONAsync(url: string): futures.Future<any> {
    var client = new httpclient.HttpClient();
    var response = await client.GetAsync(url);
    return JSON.parse(response.responseText);
}

async function fetchAuthProviders(vm: AuthViewModel): void {
    var providers = await getJSONAsync(authProviderUrl);
    vm.providers(providers);
}
```

- ``async`` keyword signals rewrite of function body
- Return type annotation must be a ``Thenable`` or ``void``
- ``void`` return type annotation signals that any errors are thrown to the browser/engine
- ``await`` keyword halts execution until the thenable produces a value or a result.
- ``await`` auto-lifts non-thenable values
- Using `async/await` requires an import of the `'generators'` module defined in `generators.ts`

Async/Await Definitions

```
interface Thenable {  
  then(resolve: Function, reject?: Function): any;  
}
```

```
interface Awaiter {  
  awaiter: Thenable;  
}
```

```
interface Deferred {  
  resolve(value: any): void;  
  reject(value: any): void;  
  promise: Thenable;  
}
```

```
declare var Awaiter: {  
  defer(): Deferred;  
}
```

- Defined in '**generators**' module
- General interface designed to match Promise/A-style promises/futures.
- Uses a built-in general purpose Promise/A-like object internally.
- Developers can replace built-in behavior by overriding the '`Awaiter.defer`' method with a compatible API (or wrapper).